Secure Multi-Player Dominoes Game

Security
Aveiro University

Daniela Lopes, Jorge Catarino,
Miguel Lopes, Óscar Pimentel

Version 1.0

Secure Multi-Player Dominoes Game

Department of Electronics, Telecommunications and Informatics
Security
Aveiro University

(85123) danielasplopes@ua.pt

(85028) jorge.catarino@ua.pt

(72385) m.lopes@ua.pt

(80247) oscarpimentel@ua.pt

January 31, 2021

Acknowledgements

We would also like to express our gratitude to our colleague Filipe Vale for providing the game base (without game accounting and any security implemented).

Also, we would like to thank the author of the book Practical Cryptography for Developers, Svetlin Nakov and the Stack Overflow Community for the various code snippets referenced in the code.

Contents

Chapter 1

Introduction

This project was developed within the scope of Security course unit.

The proposed challenge consists in the development of a system in which users can create and participate in secure online domino games. This system comprises a table manager and a bounded set of players.

To ensure an efficient security of the game, a set of features were followed. For starters, anonymity and authentication of players are assured, as well as each player's identity. Also, the distribution of the tiles is guaranteed to be random and confidential by involving all players in the stock distribution process. During the game, all kinds of cheating must be neutralized by enabling complaint mechanisms, promoting the players' honesty as well as the correct evolution of the game. Finally, the game accounting is handled, and the points are assigned to the winner.

All these features will be properly discussed and explained in the following chapters for a more in-depth understanding of the project.

Chapter 2

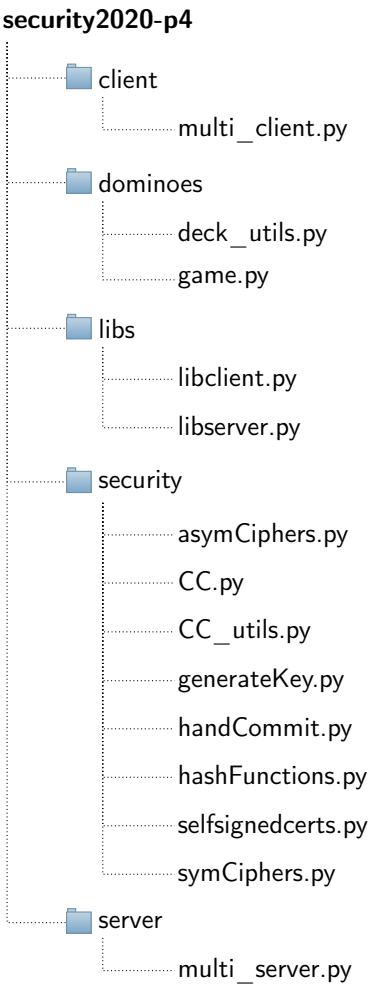
Functionalities implemented

All of the following functionalities were implemented and served as milestones during the project development.

- Protection of the messages exchanged;
- Set up sessions between players and table manager;
- Set up of sessions between players;
- Deck secure distribution protocol;
- Validation of the tiles played during a game by each player;
- Protest against cheating;
- Possibility of cheating;
- Game accounting;
- Claim of points for an identity provided by a Citizen Card;
- Picking of stock tiles during a draw game;

Chapter 3

Code Organization



File	Description
multi_client.py	Module to initialize a client and establish connection to the server
multi_server.py	Module to initialize a server and start listening for clients.
libclient.py	Module implementing message handling for the clients, including some game logic.
libserver.py	Module implementing message handling for the server, including some game logic.
asymCiphers.py	Module containing functions related to asymmetric encryption.
symCiphers.py	Module containing functions related to symmetric encryption.
CC.py	Module containing encryption function for the Citizen Card.
CC_utils.py	Module containing utility functions for Citizen Card.
generateKey.py	Module containing functions to generate random keys.
handCommit.py	Module containing functions to generate and verify hand commitments.
hashFunctions.py	Module containing fuctions for hashing.
selfsignedcerts.py	Module containing functions to generate self-signed certificates.
game.py	Module containing data structure maintaining the game information.
deck_utils.py	Module containing Player and Deck related functions and data.

Table 3.1: Description of the main files

Chapter 4

Protection

This chapter is dedicated to explain how we keep the message exchange secure. Every message contains a action parameter which the client or the server interpret to decide what to do, and a group of parameters, data that will be use in the handling of that action.

4.1 Message Encryption

For the message encryption, we mainly use *AES* in the Galois Counter Mode, with a new Initialization Vector on each use. The process used to exchange the sessions keys that permit the use of this cipher by clients and server is explained in Chapter 5.

The messages that revealed the need to be kept secret from external parties or even the server were encrypted. Communications between players during the Selection and Tile de-anonymization preparation stages (*selection_stage*, *deanon_stage*) are examples of such messages. When the server sends the translated tile after a stock pick (*insert_in_hand*), the message containing it is encrypted too.

4.2 Message Signing

For the message signing, it was decided to make use of the *RSA* protocol and the Portuguese Citizen Card capabilities. The messages that require having their integrity controlled were signed, such as:

- Messages that can alter the game flow;
- Messages that directly impact the next state;

- Messages that benefit a specific player;
- Messages that control the outcome of an action;
- Messages where authentication is needed.

These messages were signed so that when they arrive at their destiny, the receiver can verify all the relevant data integrity and continue the process, knowing that the received data was valid and by accepting it, no harm will be caused to any of the mentioned cases.

The messages signed are present in the following table.

Server		**Client**	
ACTION TO SEND	PARAMETER	ACTION TO SEND	PARAMETER
"you_host"	*"signed_session_key"*	*"req_login"*	*"signature"*
"new_player"	*"signed_session_key"*	*"aes_exchange"*	*"signed_aes_keys"*
"key_exchange"	*"signed_session_keys"*	*"send_commit"*	*"commit"*
"send_pub_keys"	*"signed_pub_keys"*	*"play_piece*	*"signed_piece"*
"reveal_everything"	*"signed_action"*	*"player_cheated"*	*"signed_player_cheated"*
"report_score"	*"signed_msg"*	*"score_report"*	*"signed_msg"*
"end_game"	*"signed_msg"*	*"validate_protest"*	*"signed_msg"*
		"validate_game"	*"signed_msg"*

Table 4.1: Relation between action sent and parameter both on server and client

Chapter 5

Communications

In this chapter, it is discussed how the communications happen between the intervening entities, as well as the security choices related to them.

Three asymmetric encryption algorithms were considered to implement the secure communications between the players and the table manager: *Diffie-Hellman*, *RSA*, and *Elliptic Curve* algorithms.

- **Diffie-Hellman algorithm:** this is a non-authenticated protocol, but does require the sharing of a *"secret"* key between the two communicating parties. This secret key is the result of the multiplication of the private number (secret) with the public number. The absence of authentication makes this algorithm vulnerable to man-in-the-middle attacks since a third party can intercept communications, appearing as a valid participant.
- **RSA algorithm:** has three main processes - key pair generation, encryption, and decryption. The key pairs are formed by a public and a private key. In this algorithm, the keys are generated by multiplying large prime numbers. Nowadays, governments and many other organizations are requiring a minimum key length of 2048-bits , since 1024-bit long keys were deemed insufficiently secure against several attacks.
- **Elliptic Curve algorithm:** can establish a level security greater or equal to RSA algorithm with less computational overhead and resorting to smaller keys. The generation of public and private keys are based on operations on points of elliptic curves.

Considering these aspects, in this project, each message exchanged is signed with *RSA (Rivest-Shamir-Adleman)*. Undoubtedly, both *RSA* and *Elliptic Curve* algorithms are better than *Diffie-Hellman*, taking into account the purposes and needs of this project. Although the *Elliptic Curve* algorithm is considered more efficient and has been growing quite promising, the choice of *RSA* lies within the fact that it is still one of the most widely used encryption algorithms worldwide, and, until now, *RSA* certificates remain unbroken.

All communications between server and clients are done with messages in *JSON* format.

5.1 Sessions Between Player and Table Manager

The player and the table manager will communicate using signed messages.

Before being able to join the table, all players must present themselves to the table manager.

The player generates a pseudonym and signs it with his citizen card private key. Then, this information is sent to the table manager, alongside the citizen card certificate, the pseudonym transformed in bytes, and the *RSA* public key.

After receiving the players' message, the table manager checks if the certificates are valid. If that is verified, the table manager uses the public key present in the certificate to verify the players' signatures. Finally, the table manager remains certain about the identity of the player and stores both the *RSA* and citizen card public keys belonging to the player.

To establish the session, the server generates an *AES-based* symmetrical session key (secret) and encrypts the symmetrical session key using the client's *RSA* public key from the previously received message, and sends it back to the player. The player uses its *RSA* private key to decrypt the server-generated session key.

Now, both player and table manager share the server-generated session key to establish a secure session with each other.

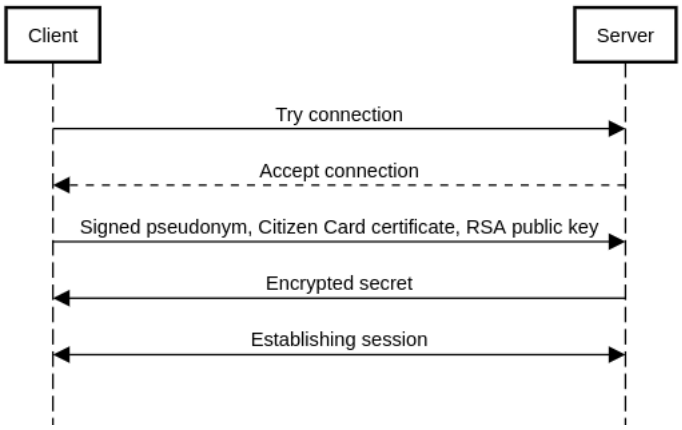


Figure 5.1: Session set-up between client and server

5.2 Sessions Between Players

The communications between players can only begin when the table is full.

To be able to communicate, all messages sent by the players must pass through the server (table manager) that forwards those messages to the corresponding receiver.

The session establishment between players follows the same method as the set up of sessions between player and table manager. An example of the session establishment between players can be observed in Figure 5.2

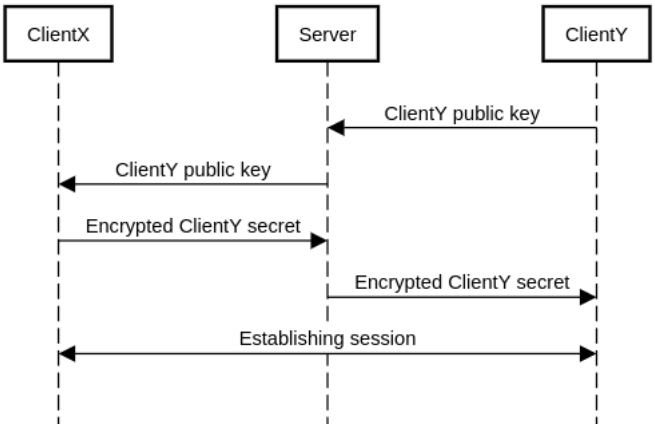


Figure 5.2: Example of session set-up between clients

Chapter 6

Deck Distribution Protocol

6.1 Pseudonymization Stage

The execution of the **start_game** action marks the beginning of the pseudonymization stage.

This stage consists of the association of the index of the tiles from the deck to a pseudonym (P_i). The pseudonym hides the real value of the tile.

To create the pseudonym, an 8-byte random alphanumeric value was generated that will serve to generate a key (tile key). This value is stored to be used later on for the de-anonymization of the associated tile. Then, a *SHA256* hash is also generated, containing the tile index (i), the tile key (K_i), and the tile itself (T_i) as inputs.

$$P_i = h(i, K_i, T_i)$$

Ultimately, the tuples containing the tile index (i) and the hash generated (P_i) are placed on a new deck that only contains anonymized tiles. This new deck is the pseudonymized deck.

6.2 Randomization Stage

In Randomization Stage the clients exchange the pseudonymized deck between them.

Each client generates a different *secret* for each tuple on the pseudonymized deck, encrypts that tuple with its *secret* and shuffles the deck before handing it over to another player.

After encrypting all the tiles (tuples), each player returns them to the deck. The client then sends the encrypted deck back to the server by performing the action **next_randomization_step**. The server will send the deck to a new player through the message **randomization_stage** until it passes throughout all the players.

After going through all players, a tuple suffered the following transformations

$$(i, T_i) \xrightarrow{K_i} \mathrm{C}_i \xrightarrow{K_i'} \mathrm{C'}_i \xrightarrow{K_i''} \mathrm{C''}_i \xrightarrow{K_i'''} \mathrm{C'''}_i$$

being C the encrypted tuple and K_i the key used to produce it (*secret*). Each player keeps a mapping of each secret it produced and the result of the encryption.

The Table Manager (server) keeps the order of the randomization stored for later use during revelation stages and stock picking during the game.

During this process, if somehow a player generates an already existing key while performing the tiles encryption, he generates a new key for that tile.

The final result constitutes the stock.

6.3 Selection Stage

In the Selection Stage, the players are secretly passing the stock between them, having two possibilities: the player draws a tile from the stock (5% chance) or let it pass to another player.

If a player chooses to draw a tile, a padding with the same size of the tile, in bytes, is added to the stock to increase the difficulty of tracing the withdrawn tiles since the message size is kept intact. Then, the player shuffles the remaining stock and forwards it to another player (this message is encrypted as described in Section 4.1 and sent to the server, with the action **send_to_player**, which only contains the encrypted message and the pseudonym of the player the server should redirect it to)

However, if a player chooses to pass the stock, he is given two possibilities: either he chooses to swap a piece from his hand or immediately sends the stock to another player. Either choice has a 50% probability of happening.

The selection stage ends when the number of tiles in the remaining stock is lower than the number of tiles present in all the players' hands. In other words, the selection stage ends when all players have their hands complete.

6.4 Commitment Stage

The hand commitment is a computation performed over the set of tiles that form the hand that can be revealed before the beginning of the game without revealing the tiles.

The commitment is calculated using a one-way *SHA256* hash (h) and two 32-bytes long random values (R_1 and R_2). Considering T has the value to commit, then the hand commitment(hc) can be computed as

$$hc = h(R_1, R_2, T)$$

After the end of the Selection Stage, the server sends a message with the action **commit_hand**. This will prompt the players to generate their hand commitments and send $(R1, b)$ to the server (table manager), on a message with the action **send_commit**.

Before sending the hand commitment, each player must sign the message to ensure the integrity of its content. Then, the server validates everyone's signatures and sends them back to the players. Each player verifies the signature of the commit thus validating the game starting point by everyone.

Once this stage is done, the Revelation Stage begins.

6.5 Revelation Stage

This stage begins when the players receive the message **reveal_key**.

The last player to encrypt the deck will start the Revelation stage. On this stage, the player will start by revealing all of his corresponding keys, that can decrypt every tile, except the ones present in the stock. When finishing this phase, the client (player) will send a message **revealed_keys** signaling the end of the reveal process for the player. All the players will receive the keys and after a round of decryption, they will then send a message with the action **waiting_for_keys** to the server, signalling they are ready to receive the next keys.

This process above will be repeated, following the randomization order, until all players have revealed their keys to everyone.

After every key is revealed, each players will decrypt the tiles, and have their corresponding pseudonomyzed hand available.

After the end of this phase, the game will start the De-anonymization Stage.

6.6 Tile De-anonymization

6.6.1 Tile De-anonymization Preparation Stage

After the conclusion of the Revelation stage, the server will send a message with the action **start_deanon_stage** to a randomly selected player. This message contains a padding list, filled with randomly generated strings with a size of the *RSA* Public Key in *PEM* encoding and a public key list filled with *None* elements, both with the size of the game deck.

In this stage, the player will have two possibilities: either he puts a key on the public key list (5% probability) or he passes to another player. In case the player chooses to place a key, he removes one element of the padding and adds a *None* to the list, thus maintaining the size of the messages being sent.

The players send this list and the padding (encrypted, as described in 4.1) with the action **deanon_stage** to the other players. When all players have inserted their public keys, one will message the server with the action **deanon_prep_over**, containing the filled public key list, so the Tile De-anonymization stage can start for real.

6.6.2 Tile De-anonymization Stage

In this stage, the table manager (server) uses the information received in the Revelation Stage and the public key list from the preparation stage. Each tile is translated and then encrypted with the corresponding public key from the list. Afterwards, it sends a message to all players with action **decipher_tiles**, containing the list of tuples with the encrypted tiles and their key (K_i).

When receiving this list, the players will pick the ciphertexts corresponding to their tiles and decrypt them. After, the player will verify the *SHA256* digest of each tile, to check if the server cheated while providing the information. If this verification is successful, the player will finally have his translated hand. To signal he is ready to play, he sends a message with the aptly named action **ready_to_play**.

After all players send this confirmation, the game will start.

6.7 Stock use

When a player has no tiles to play, and the stock is not empty, he has the option to pick a tile. To do this he removes a tile from the stock and send the server a message with the action **request_piece_reveal**, with the tile he desires to pick and the new updated stock.

After receiving this, the server will use the randomization order to get the keys to decrypt that tile from all the players, sending each of them messages with the action **reveal_piece_key**. Each of the players will then answer with another message with the action **revealed_key_for_piece**, containing the key necessary to decrypt the tiles. This process is similar to what is done in the Revelation Stage.

Receiving these keys and having the pseudonymized tile, the client will ask the server for a tile de-anonymization, using a message with the action **request_piece_deanon** containing the pseudonymized tile. The table manager will then translate the tile and send a encrypted message to the client containing it and the tile key.

The player will then check the *SHA256* digest of the tile, to check if the server did not cheat while providing the information. Afterwards, he inserts it in his hand and continues playing.

Chapter 7

Game Process

This chapter describes the game logic implemented, touching the game rules, how a game flows and is accounted and how cheating can happen and be detected.

7.1 Game Rules

- The deck consists of 28 tiles;
- The numbers of players may vary between 2 and 4 players;
- Each player's starting hand consists of 5 tiles. However, if there are only 2 players, the starting hand is formed by 7 tiles.
- To build their hand, the players rotate the stock between them and withdraw a tile(This is accomplished with the Deck Distribution Protocol described in Chapter 6);
- The first to play is the player that arrives first at the table;
- In order to play, each player must match the pips on one end of a tile from his hand with the pips on an open end of any tile in the domino chain;
- When a player does not have a valid tile to place, he must pick one or more tiles from the stock to add to his hand until he is able to make a valid play;
- If there are no tiles to pick from the stock, the player should pass the turn;
- Whenever a player is caught cheating, the game ends;
- At the end of the game, the winner earns the sum of the numbers present in each opponents' remaining tiles.

7.2 Play a Game

After joining a table, the player has to wait until the table is full to start the game.

The game will start automatically, without needing human intervention, after performing the deck distribution protocol processes.

During the game, both the domino chain and the hand of the player are updated whenever a player places a new tile. A player can play legally or can cheat tiles.

7.3 Cheating

A player can cheat by playing a piece that he doesn't have in his current hand and he didn't pick from the deck.

There are two cheating options: automatic and manual. The user is asked if it wants to activate the manual cheating. If it does activate the manual cheating, the user can choose manually the values for the cheated piece.

It was decided that the automatic cheating function would work similarly to the normal play function, with the difference being when the player doesn't have a valid play in his hand he doesn't pass or pick a piece from the deck. The player checks the table edges and creates a new piece with those edges, guaranteeing that the cheated piece is always a playable piece. The player erases a piece from his hand, creates the cheated piece with the edges' numbers, adds it to his hand and plays it, like if it was in his hand from the start.

7.4 Cheating Detection and Protest

The clients (players) check if the played piece was already in the table and everyone checks if it was a piece that matches one in their hands. This way, if a player cheats a piece that matches either one of the table or one in another player's hand, it is immediately caught and every other player can protest that play.

If a player protests, the server checks the suspected cheater's bit commitment. If the piece was not in that bit commitment, the server checks if that piece's cipher key was already used. If any of these checks fail, it means the player played a piece that was not in his initial hand or part of his actual hand, and the cheater is detected, therefore ending the game. Since everything has to be revealed so the table manager can handle a cheating protest, even if the player is not a cheater, the game will end.

The server plays a part on the cheating detection too. If the table manager detects that a player used a piece that wasn't already translated or is playing a duplicate, he will start a a cheating validation process.

At the end of the game, the Table Manager does a full game validation, where it verifies every player hand commit, what they played during their game and their remaining hand, so if any cheaters weren't caught before, they are caught now.

7.5 Game Accounting

The accounting process occurs at the end of the game, which is achieved when a player comes out victorious, or a tie occurs.

A player comes out victorious when he is the first to run out of tiles in his hand or, in case of a draw, he is the one ending up with the fewer points in his hand. A draw occurs when a game runs into a deadlock due to the lack of valid tiles to play.

When any of the above situations occur, the table manager sends the remaining tiles to all players - the table manager has previously acquired the tiles while performing the game validation - so they can calculate for themselves the score to be awarded to the winning player. At the same time, the table manager performs these calculations as well to compare with the results obtained by players. If they all agree, the server will proceed to assign the score.

To assign the score, the table manager launches a Challenge (final score) to the winning player. The player responds with the Challenge signed with his Citizen Card private key, and the table manager decrypts the message with the player public key - previously acquired while establishing the initial session (view section 5.1).

Finally, the table manager compares the score value from the message he sent with the one he received and decrypted from the player. If they match, the players' identity is confirmed, and the score is assigned to him. If the player's certificate is invalid, due to expiration or any other reason, the points are not assigned.

The final result is stored in a file together with the pseudonym of the winning player.

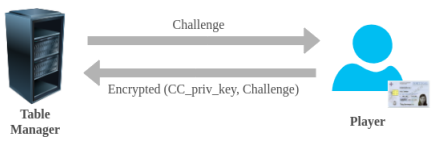


Figure 7.1: Challenge-response approach

Chapter 8

Observations

Some problems and deficiencies were identified.

- Without some delay before writing a message to a socket, sometimes a message is lost;
- If another player tries to enter in a table during a game, when it is already full, during certain phases, the game might crash;
- We believe the security in the initial exchange of public keys could be improved, with the use of public key certificates, with the Table Manager taking the role of a CA (Certificate Authority).

Chapter 9

User Manual

There are two main components: the server and the multiple clients.

First of all, in order to be able to run the game, the user must install the requirements needed. To do so, run the following command

```
$ pip install -r requirements.txt
```

The server, which serves as table manager, only delivers a game at a time. To start this component, execute the following command

```
$ python3 multi_server.py
```

alongside the following arguments

```
OPTIONS:
  -i,  --ip           Set the server IP address
  -p,  --port         Set the server port
  -np, --num_players  Set the maximum number of players [OPTIONAL]
```

The multiple players are simulated using a single client, which can be run by each player. To start the clients, run the following command in different terminals (as many as the number of players)

```
$ python3 multi_client.py
```

alongside the following arguments

```
OPTIONS:
  -i,  --ip           Set the client IP address
  -p,  --port         Set the client port
  -c, --cheater       Set the player as a cheater. Default:False
                      [OPTIONAL]
```

NOTE: If Python is having trouble finding the modules, try commenting the following lines on *multi_server.py*, *multi_client.py*, *libserver.py* and *libclient.py*.

```
sys.path.append(os.path.abspath(os.path.join('.')))
sys.path.append(os.path.abspath(os.path.join('..')))
```

Chapter 10

Conclusions

This project provided many interesting discussions and decisions about security, as exemplified by the choices taken for the communication messages, that led to a further research about encryption algorithms and to the deepening of knowledge about security concepts like symmetric and asymmetric ciphers and signings.

The project also made possible to better understand how to build a secure client-server system with integrity guarantee on its content.

We recognize that developing any secure game or application requires a very meticulous working method, as well as good programming practices and knowledge of cryptographic methods.